Parallel, multigrain iterative solvers for hiding network latencies on MPP's and networks of clusters *

James R. McCombs † Andreas Stathopoulos †

January 15, 2003

Abstract

Parallel iterative solvers are often the only means of solving large linear systems and eigenproblems. However, these solvers are usually implemented in a fine grain manner and, when scaled to large numbers of processors on MPP's, can incur significant performance penalties due to synchronization overheads. This problem is exacerbated in clusters of workstations (COWs) and SMPs that are interconnected via a hierarchy of commodity networking components using standard communication protocols. Because overheads in MPPs and LAN technologies have not improved nearly as much as network bandwidth in recent years, there is a need for innovative parallel implementations of scientific applications that are capable of hiding overheads. In this paper, we describe a novel scheme for improving the scalability of a particular class of numerical algorithms, specifically, by hiding the overheads of block iterative solvers that employ flexible preconditioning through an inner iterative method.

Block methods are not only robust in the presence of eigenvalue multiplicities and multiple right-hand sides, but provide better latency tolerance by performing more floating-point operations between synchronizations. We take a different approach to inducing latency tolerance by increasing the granularity at which the preconditioning is performed for each block vector. This is accomplished by splitting the processors into smaller subgroups which are then used to precondition each block vector concurrently. The rest of the algorithm is still performed in fine-grain. We call this combination of fine and coarse-grain parallelism *multigrain*.

To test the effectiveness of the multigrain parallelism, we implemented a multigrain, block Jacobi-Davidson algorithm for computing a few extreme eigenvalues of a symmetric matrix. We obtained improvements of 45-50% over both the block and non-block implementations of the fine-grain method when testing on an IBM SP and on a collection of clusters consisting of Sun workstations.

1 Introduction

Many applications in science and engineering require the solution of linear systems of equations or the computation of a few extreme eigenvalues of a large, sparse matrix A. Iterative methods are often the only way of solving these problems. GMRES [21] for linear systems and Arnoldi [6] for symmetric eigenvalue problems are two popular choices. Preconditioning can be applied to accelerate convergence when the matrix is ill-conditioned in the case of linear systems [25], or its eigenvalues are tightly clustered in the case

* Work supported by the National Science Foundation (ITR/ACS-0082094 and ITR/AP-0112727), and performed using computational facilities at the National Energy Research Supercomputing Center and the College of William and Mary (National Science Foundation [EIA-9977030] and Sun Microsystems [SAR EDU00-03-793]).

† Department of Computer Science, College of William and Mary, Williamsburg, Virginia 23187-8795, (mccombjr/andreas@cs.wm.edu).

of eigenproblems [20]. Two popular variants of GMRES and Arnoldi, FGMRES [24] and Jacobi-Davidson [28], are methods that allow for flexible preconditioning. Block algorithms may be used to further accelerate convergence on linear systems with multiple right-hand sides [18] and eigenproblems with highly clustered or multiple eigenvalues [12]. They are also more robust because single-vector methods can misconverge to unwanted eigenvalues in the presence of eigenvalue multiplicities. Although block methods improve cache performance, the number of matrix-vector operations (and thus flops) performed is usually higher than their single-vector counterparts. Hybrid techniques have been developed that attempt to keep the number of matrix-vector multiplications from increasing excessively [26]. In our approach, we use the original block approach and take advantage of its coarse-grain parallelism to tolerate high overheads despite a slight increase in the number of matrix-vector operations.

Most iterative methods are implemented in a data-parallel (fine-grain) manner [9] that requires several synchronization points at each iteration and is more suitable for MPP's, Global reductions, a common reason for synchronization in linear algebra codes, are particularly expensive because the data being exchanged between processors is small compared to the overheads. The synchronization costs become even greater in collections of clusters and Grids where the networking resources incur a higher latency and are shared by multiple parallel jobs competing for the network. Yet, commodity components and high-speed networking media make these environments an increasingly cost effective option for scientific computing [14]. These environments can consist of a single cluster of workstations (COW) or a collection of COW's interconnected via a hierarchy of switches. The result is a heterogeneous networking environment where not all nodes incur the same latency to communicate with each other. There is also a trend toward Grid environments where the clusters may be geographically dispersed and interconnected by a high-latency WAN [11].

Reductions in overheads have not kept up with increasing network bandwidth and processor speeds – a trend that seems will continue. Although this is true for MPPs, it is even more apparent in low-end parallel environments such as COWs. For instance, Fast Ethernet and Gigabit Ethernet have estimated combined hardware and software overheads of about 440μsec and 300μsec respectively and the IBM SP has 39μsec [13]. Even though recent versions of Myrinet with expensive proprietary network interface cards claim latencies of under 10μsec, these timings do not include overheads from message passing libraries such as MPI. As hardware doesn't seem to provide a solution to the latency problem, we turn to algorithmic design.

Because of the sequential nature of iterative methods, it is difficult to incorporate coarse-grain parallelism into them. Related research has focused mainly on either reducing the number of synchronization points per iteration, or by introducing more work between reductions through block methods [8, 3, 19, 29], but it hardly addresses the above overhead problems, especially on COWs and heterogeneous networks. A completely coarse-grain implementation was discussed in [19] where processors store entire columns of the basis vectors but the amount of interprocessor communication during the orthogonalization and projection phases is unreasonably high.

In [30], we introduced a coarse-grain Jacobi-Davidson solver that lets each processor apply the preconditioning to a distinct block vector independently. This allowed us to arbitrarily increase the computation/communication ratio by increasing the accuracy of the inner solve. However, this implementation was restrictive because it required the block size be a multiple of the number of processors and that each processor store the whole matrix. We eliminated these restrictions in [17] by splitting the processors into k, the block size, subgroups that perform the preconditioning of each block vector concurrently. Within the individual subgroups, the inner solve is implemented in a fine-grain way. The rest of the solver is performed in fine grain. We refer to this combination of fine and coarse-grain parallelism as *multigrain*. For a fixed problem size, the speedup of the fine-grain solver implementation begins to decrease beyond a certain number of processors P_{opt}. The purpose of multigrain is to extend P_{opt} by as much as a factor of k. To reduce inter-cluster communication in collections of clusters, we define the subgroups so that they correspond to separate COWs or subsets of processors within the same COW. Given the abundance of RAM in modern workstations and MPP's, this does not place an unreasonable burden on memory resources since each processor must only store an additional N/g rows of the coarse-grain partitioning where P is the total number of processors and g is the size of each subgroup. This is a reasonable requirement given the low cost of RAM and the fact that workstations now support up to a gigabyte or more of RAM.

In this paper we first describe a class of iterative methods for which the multigrain technique is applicable — block methods formulated as an inner-outer iteration. We then describe the implementation of the multigrain technique and its variations for use in cluster environments and on MPPs. We build on previous work by extending the capabilities of our multigrain library to work efficiently on MPP's that are clusters of SMP's (such as the IBM SP) and more traditional MPP's such as Crays. To show the benefits of these extended capabilities, we conducted new tests with our block Jacobi-Davidson solver with multigrain capabilities on an IBM SP and on a collection of clusters consisting of Sun workstations. Finally, we construct a model that describes under what conditions multigrain is beneficial.

2 A multigrain paradigm for preconditioned, block iterative methods

Many scientific and engineering applications involve the solution of a system of linear equations $A\tilde{x} = b$ for the unknown vector $\tilde{x}$ or the solution of the eigenvalue problem $A\tilde{x}_i = \tilde{\lambda}_i\tilde{x}_i$, for the $i = 1, \ldots l$ smallest or largest eigenvalues $\tilde{\lambda}_i$ and the corresponding eigenvectors $\tilde{x}_i$. When the matrix A is large and sparse, iterative methods provide the only means of solving these problems. Our research is focused on a class of iterative methods that have an inner-outer iterative structure where the inner iterations are used to precondition the linear system.

Many block iterative methods such as block GMRES for linear systems and block Jacobi-Davidson for eigenvalue problems follow this structure. Considering a block size of k, these methods build a subspace V from where they extract their approximate solutions. At each iteration, they build the next k vectors by approximating solutions to k different correction equations, one per block vector. The approximate solutions are then orthogonalized and appended to the subspace V. This process is illustrated in Figure 1. The correction equations are usually solved using m steps of another preconditioned iterative solver such as CG or BCGSTAB. It is preferable to employ an inner solver that depends on short recurrences because they require a fixed amount of computation per iteration and only a few vectors to store. The more accurately the inner systems are solved the fewer outer iterations the algorithm performs. However, there is usually an optimal number of inner iterations beyond which the actual time, as measured by the total number of matrix-vector multiplications, slowly increases.

Despite improved cache efficiency and a relatively coarser granularity in data-parallel implementations, block algorithms are only competitive when access to the matrix is expensive so that it is beneficial to block several matrix-vector operations per matrix access. Instead of increasing the amount of work performed during matrix-vector operations, as has been the case with block methods, multigrain increases the granularity of the matrix-vector and dot product operations during the correction equation. We note that the correction equations are independent for each of the block vectors and each may take an arbitrarily long amount of time. If we assign each correction equation to a different subgroup of processors, each subgroup should be able to perform the majority of its computation without communicating with other groups. Thus, for subgroups of equal size, we effectively reduce the latencies in our parallel computer to the latencies of a computer with $1/k$th the number of processors. As a result, the multigrain solver can be scaled to k times as many processors as the fine-grain solver. This is particularly beneficial when each subgroup represents

```
while ( x_1 ... x_k are unconverged) {
    if (solving linear systems)
        Set s_i = v_i, for i = 1, ..., k
    elseif (solving eigenvalue problem)
        Set s_i = x_i, for i = 1, ..., k
    endif
    Apply m inner iterations on s_i to obtain t_i
    Append t_i to V and orthogonalize V
    Compute the new approximations x_1 ... x_k
}
```

Figure 1: Inner-outer methods with this structure can be implemented with multigrain parallelism. The inner iterations are often themselves a preconditioned iterative solver. The v_i are the column vectors of the basis V and k is the block size

a COW with high performance intracluster, but not intercluster networks. Note that each subgroup (COW) should be able to keep the whole matrix A to solve the correction equation. The additional memory burden is usually affordable since each processor needs to store only k times more rows than its fine grain partition. For example, multigrain on a relatively dense sparse matrix of 100 non-zero elements per row and $128,000$ local rows, would require 512MB of RAM – a small amount by todays standards. If 100 processors are used, the solvable problem size would be 12.8 million unknowns!

Except the correction equation, the rest of the steps of the algorithm in Figure 1 (orthogonalization, computation of new approximations, etc.) cannot be performed efficiently in the above coarse-grain setting. Fortunately, they comprise only a small portion of the total execution time. Thus, multigrain follows the traditional fine-grain partitioning for all the other steps, and switches to coarse grain only for the correction phase. An all-to-all operation is required to transition each of the fine grain vectors s_i and t_i to their coarse-partitioned counterparts on each processor, so that each subgroup has its respective vector. We refer to this first all-to-all as an `MG_Gather` operation. Despite the high cost of the all-to-all operation, the number of inner iterations (and thus granularity) can be increased arbitrarily to diminish the associated latencies. This is reasonable for difficult problems where several inner iterations must be applied. After the inner solve is complete, a second all-to-all operation, an `MG_Scatter`, is needed to transition the coarse t_i from the coarse-grain partitioning back to their original fine-grain partitioning among all processors.

2.1 Partitioning schemes

The complexity of the all-to-alls depends on how the fine and coarse-grain partitionings are constructed. If the two partitionings are independent, the all-to-alls must involve all processors. The overheads in a total exchange between all processors can greatly impede the performance of the multigrain solver. To reduce the all-to-all time, we construct a hierarchical partitioning where the fine-grain matrix is derived by partitioning the coarse-grain matrix. Before we go into detail of how this is done we next describe the method of independent partitionings.

2.1.1 Multigrain algorithm using independent partitionings

In the most general case, a single fine-grain and multiple coarse-grain partitionings, one for each subgroup, are computed separately from each other using some partitioning software [22, 15]. Thus, no assumptions can be made about which coarse-grain rows of the vector s_i a processor will need to acquire before executing the inner-solve. For instance, processor 0 may have rows 2, 5, and 8 of the fine-grain matrix, but may require rows 0, 5, 7, 12, 16, and 30 of the coarse-grain matrix. The rows a processor requires may reside on several different processors, increasing the amount of communication and requiring that all of the processors be involved in the all-to-alls.

Despite the high cost of all-to-alls in this case, this method of independent partitionings is useful on heterogeneous collections of clusters where each cluster may be of different size, and thus there is no simple matching between fine and coarse-grain partitions. Because heterogeneity of subgroups, henceforth referred to as *solve groups*, load imbalance could occur during the preconditioning phase. In [16] an application-level load balancing scheme which forces all solve groups to spend the same amount of time in the inner solve regardless of how few iterations each performs. In this paper, we compare the global all-to-all performed by the method of independent partitionings with the more efficient method of hierarchical partitionings discussed below.

2.1.2 Multigrain algorithm using a hierarchical partitioning

When the solve groups are relatively homogeneous and of equal size, we can significantly reduce the all-to-all time, by considering the following hierarchical partitioning of the matrix. Assume for simplicity that the block size, k, divides the number of processors, P. First, we obtain the coarse-grain partitioning of the matrix onto P/k processors. This will be used by each of the solve groups during the correction phase. Each of the resulting P/k coarse-grain partitions are then partitioned further into k pieces by applying the partitioner to the square diagonal block which is roughly of size $N/(P/k)$. This yields each processor's local fine-grain partition. Now P/k groups can perform their all-to-all operations simultaneously. These groups, which we refer to as all-to-all groups, are only of size k processors each — significantly decreasing the amount of traffic and overhead involved with the all-to-all operations.

Figure 2 illustrates the behavior of multigrain with hierarchical partitionings through an example with $P = 6$ and $k = 2$. Before the correction phase, each member of an all-to-all group sends its fine-grain portions of the k vectors $\mathbf{s}_i$ to the members of its solve group, and receives the pieces that compose its coarse-grain portion of $\mathbf{s}_i$. After each solve group finishes its inner iterations, the all-to-all is reversed and each processor's coarse-grain portion of $\mathbf{t}_i$ is distributed across all the processors in the all-to-all group.

2.2 Process-to-node mapping

Processes should be assigned to solve groups depending on the target parallel environment: MPP of SMP's, COW, or collection of clusters. We describe this mapping on three typical parallel environments.

We first describe the simplest environment, a traditional MPP such as a Cray T3E capable of running SPMD programs. In these environments, users often do not have control over and cannot even discover which processors are allocated to their job. Once the multigrain library is initialized by the application at run time, the processes are assigned fine and coarse-grain rows of the vectors and matrix according to the default mapping in Figure 2.

Second, on MPPs that are clusters of SMP's such as the IBM SP, the process mapping becomes more complicated. As in the case of traditional MPP's, the user has little or no control over which nodes are

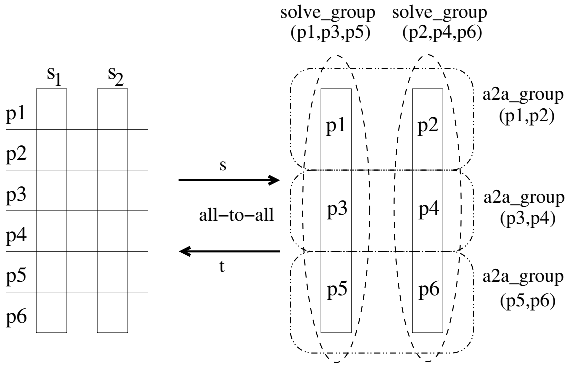


Figure 2: Example of multigrain with hierarchical partitioning with six processors ($P = 6$) and two block vectors/solve groups ($k = 2$). Before the preconditioning phase, nodes in the same all-to-all group receive the coarse-grain portions of the vectors s_i they are responsible for. Each solve group then performs its respective preconditioning. Afterwards, each node scatters its coarse-grain portion of $\mathbf{t}_i$ among its fellow all-to-all members.

allocated. However, the application can control how SMP's are assigned to solve groups. We would like processes on the same SMP to be members of the same solve group so that, during the correction phase, communications between processes of another solve group do not interfere with the traffic on this node. We have implemented a mapping that guarantees solve groups to consist of collections of SMP's and that no two solve groups share the same SMP. We accomplish this during initialization of the multigrain library by having each process call the Unix system call `gethostname`, and then binding processes with the same host name to the same solve group. This grouping of SMP's also facilitates nearest neighbor communication and utilization of shared memory during the correction phase by placing processes with contiguous MPI solve group ranks on the same SMP.

Collections of clusters are the most general case. Due to heterogeneity in the network, a user needs to be able to specify which machines will comprise each solve group. Therefore, we assume that the batch scheduler, if one is being used, allows users to request machines by name. Given this assumption, we explicitly map processes to nodes using a solve group configuration file that gives the names of the machines to be included in each solve group. If more than one process is running on an SMP, then both processes are made members of the solve group with which the SMP is associated.

3 A multigrain, block Jacobi-Davidson implementation

Jacobi-Davidson is an algorithm for computing extreme eigenvalues of large, sparse matrices and is best suited for difficult eigenproblems where the desired eigenvalues are clustered together. During the correction phase, it performs a substantial amount of work to compute corrections to the approximate eigenvectors, and thus it is a natural candidate for multigrain.

At each iteration, the method computes the current approximate eigenvalues λ_i, $i = 1 \dots k$, referred to as Ritz values, and the approximate eigenvectors x_i, referred to as Ritz vectors, using the Rayleigh-Ritz procedure. The associated residuals $r_i = Ax_i - \lambda x_i$ are then computed and used to solve k correction equations,

$$(I - x_i x_i^T)(A - \sigma_i I)(I - x_i x_i^T) t_i = r_i, \tag{1}$$

for the vectors t_i, which are approximations to the error in x_i. Usually $\sigma_i = \lambda_i$ in the correction equation, but it can also be chosen so that it is close to the eigenvalues that the user wants to compute. The corrections are then orthogonalized against the basis V before being appended to V. Extension of V using the corrections improves the accuracy of the approximations in future iterations. An outline of a fine grain, block JD algorithm is given below.

Algorithm: **Block JD**
starting with k trial vectors t_i
While not converged do:
1. Orthogonalize $t_i, i = 1 : k$. Add them to V
2. **Matrix-vector** $W_i = AV_i,\ i = 1 : k$
3. $H = V^T W$ (local contributions)
4. Global_Sum(H) over all processors.
5. Solve $Hy_i = \lambda_i y_i,\ i = 1 : k$ (all procs)
6. $x_i = V y_i,\ z_i = W y_i, i = 1 : k$ (local rows)
7. $r_i = z_i - \lambda_i x_i,\ i = 1 : k$ (local rows)
8. **Correction equation** Solve eq. (1) for each t_i

end while

Steps 1-7, referred to as the *projection phase*, perform the orthogonalization, extension of V, and the Rayleigh-Ritz projection. During the correction/preconditioning phase, k different equations (1) are solved approximately for the t_i, usually by employing an iterative solver for linear systems such as BCGSTAB or CG [25] that make use of short recurrences. Preconditioners such as sparse approximate inverse or incomplete LU factorization may be used to accelerate the convergence of BCGSTAB. When more that k eigenvalues are to be computed, the implementation replaces a converged x_i with one of the remaining unconverged Ritz vectors and continues.

3.1 Multigrain Jacobi-Davidson implementation

The projection phase of the algorithm remains the same in the multigrain modification, but step 8 must be expanded by calling `MG_Gather` and `MG_Scatter` before and after the correction equation as was shown in Figure 2. Before the correction phase, each solve group must obtain its respective coarse-grain x_i and r_i vectors. The preconditioned inner solver is then applied by each solve group to solve for a different correction vector using the technique described in [27] for applying a preconditioner of A to the projected matrix in 1. For the difficult problems in our experiments, 10-20 iterations of the inner solver are usually needed to obtain timely convergence of the target eigenvectors. After the corrections have been computed they must be transitioned back to fine grain before they can be orthogonalized and appended to V. The required modification to step 8 is given in Figure 3.

8. **Multigrain correction phase in Jacobi-Davidson**

`MG_Gather`:

  send local fine-grain rows of x_i, r_i to each solve group i

  receive coarse-grain rows for $x_{mygroup}, r_{mygroup}$ from each proc

Apply m steps of (preconditioned) BCGSTAB on

  eq.(1) with the gathered $r_{mygroup}$, $x_{mygroup}$

`MG_Scatter`:

  send coarse-grain rows of $t_{mygroup}$ to each proc

  receive fine-grain rows for t_i from solve group i

Figure 3: The multigrain modification of step 8 in the block JD algorithm.

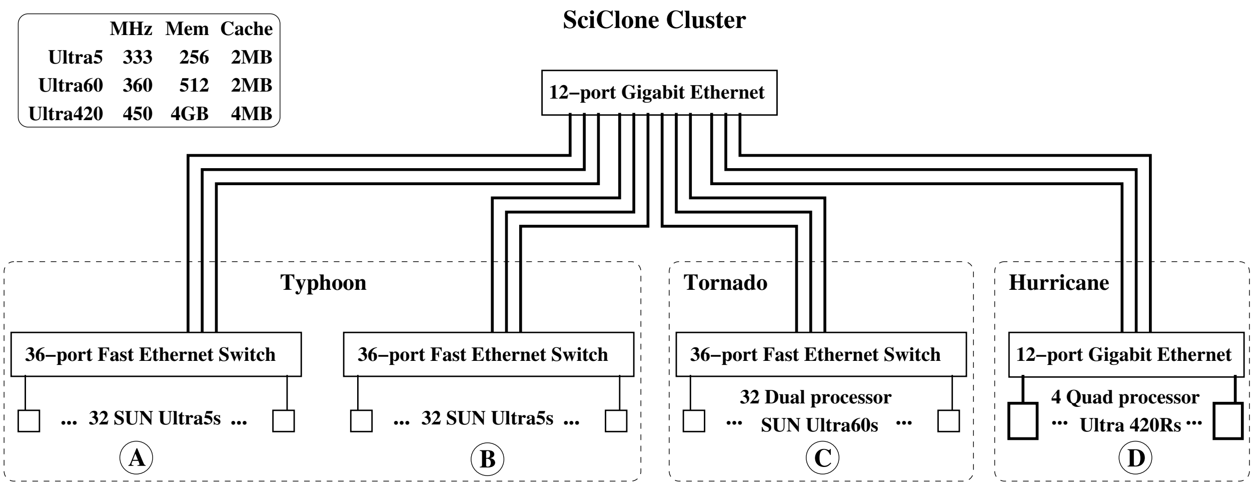


Figure 4: SciClone: The William and Mary heterogeneous cluster of three homogeneous clusters: Typhoon, Tornado (also called C), and Hurricane (also called D). We distinguish between A and B, the subclusters of Typhoon, because their intercommunication passes through the Gigabit switch.

Matrix	Dimension	Non Zeros	m_{max}	SPAI Parameters Level	Threshold
FL3D268	$268,515$	$3,926,823$	20	2	0.10
CFD2	$123,440$	$3,087,898$	10	1	0.05
NASASRB	$54,870$	$2,677,324$	10	2	0.05

Table 1: The dimension, number of non zero elements, maximum number of inner iterations m_{max}, and the preconditioning parameters are given for each test matrix. The level and threshold control the density of the preconditioner. Higher values for `level` result in denser preconditioners while higher values of `threshold` result in sparser preconditioners. Recommended values for `thresh` $\in [0.01, 0.10]$ [5].

4 Experiments

We conduct experiments with the fine-grain and multigrain Jacobi-Davidson code on two parallel architectures: a collection of clusters consisting of Sun workstations and an IBM SP. The workstation clusters, collectively referred to as SciClone, are used for scientific and parallel computing research at the College of William and Mary. It is a suitable environment for experiments in cluster computing because it is composed of heterogeneous LAN technologies and is organized as a hierarchy of switches linking various COWs (Figure 4). The IBM SP uses a single proprietary low-latency switching technology to interconnect SMP nodes of 16 processors each [2, 1].

4.1 Test matrices and solver parameters

We perform tests using three matrices. The first matrix, FL3D268, is derived from a finite-element problem [4]. The second matrix, CFD2 [7], is derived from computational fluid dynamics. The third matrix, NASASRB [7], is a much smaller but more dense matrix. These matrices are chosen because their corresponding eigenproblems are difficult to solve and require a significant number of inner iterations for the algorithm to converge in a timely manner. We use BCGSTAB from SPARSKIT [23] as the inner solver, preconditioned by the ParaSails sparse approximate inverse (SPAI) library [5]. Table 1 gives the sizes of each matrix and the parameters used to generate the preconditioners. A shift of $\sigma = 0$ is used because each matrix is symmetric positive definite and we target the lower end of the spectrum. For each test, 50 eigenvalues are computed and a maximum basis size of 108 with a restart size of 60 vectors is used. The performance of the fine-grain and multigrain codes are compared using both block sizes $k = 1$ and $k = 4$. The multigrain experiments are tested using two and four solve groups. When only two solve groups are created, each solve group solves two correction equations for every outer iteration (for two out of the four block vectors).

Approximate eigenvalues are considered converged once the residual norm is below $10^{-15}||A||_F$ for matrices CFD2 and FL3D268, and $10^{-11}||A||_F$ for NASASRB. The inner solver is given a zero vector as initial guess and iterates for m_{max} steps (see Table 1) or until q_m, the residual of the inner system, satisfies:

$$q_m \leq base^{-iter}||q_0|| + \lambda_{max}\epsilon_{mach}, \tag{2}$$

where q_0 is the initial residual of system (1), $iter$ is the number of outer iterations performed by JD thus far [10], λ_{max} is the maximum Ritz value computed thus far, and ϵ_{mach} is machine epsilon. We use λ_{max} as an approximation tor$\|A\|_2$. For CFD2 and FL3D268, $base = 1.05$, and for NASASRB, $base = 0.8$. All the above parameters are chosen because they result in the fastest execution of the fine-grain code.

Independent partitionings

Configuration	$A_{16}A_{16}C_{16}C_{16}$ - 64 processors				$A_{32}B_{32}C_{32}C_{32}$ - 128 processors			
Matrix	Gather Scatter	MG inner solve	MG Total	FG	Gather Scatter	MG inner solve	MG Total	FG
FL3D268	0.212	3.05	3.26	5.28	1.820	2.27	4.09	5.27
CFD2	0.176	2.58	2.76	3.61	0.319	1.79	2.11	3.11
NASASRB	0.071	1.62	1.69	1.81	0.770	1.04	1.81	1.60

Figure 5: Timings for one correction phase for fine-grain (FG) and multigrain (MG Total) Jacobi-Davidson, using the method of independent partitionings on 64 and 128 processors, with 4 solve groups and $k = 4$. We also show the times for multigrain to perform both the `MG_Gather` and `MG_Scatter`, and for solve groups to execute m_{max} iterations of BCGSTAB (MG inner solve). The configuration string "$A_{16}A_{16}C_{16}C_{16}$" indicates that four solve groups of 16 processors each were used – two groups of size 16 from cluster A and two groups of size 16 from cluster C. A similar string describes the 128-processor configuration.

4.2 SciClone tests

We perform several fine-grain experiments by using various configurations from 16 up to 128 processors on the typhoon and tornado subclusters. Tests on the typhoon subcluster requiring 32 or fewer processors can be performed without communication traveling across more than one switch. Because contention for the Ethernet interface can occur when more than one processor is utilized on the Ultra 60's, tests requiring only 16 or 32 processors on the tornado subcluster are run using one processor per node. All tests were performed using a shared memory version of LAM MPI and Fast Ethernet.

4.2.1 Independent versus hierarchical partitioning

The method of independent partitionings of Section 2.1.1 is useful in heterogeneous clusters, but it incurs high overhead because the all-to-all operations are between all processors. Our hierarchical partitioning limits the all-to-all operations to groups of processors of size equal to the number of solve groups. We have recorded the amount of time spent performing the `MG_Gather` and `MG_Scatter` operations and the amount of time spent executing m_{max} iterations of BCGSTAB. Tables 5 and 6 show the results for each matrix and each partitioning scheme, for 64 and 128 processors, a block size of $k = 4$, and four solve groups. The two methods are competitive for 64 nodes, but the scatter and gather operations become too expensive when independent partitionings are used with 128 nodes. These operations are significantly faster under the hierarchical partitioning because data is exchanged within all-to-all groups of size 4. The remainder of the performance tests described in this paper are performed using the hierarchical partitioning.

4.2.2 Fine-grain versus multigrain Jacobi-Davidson

We test the fine-grain and multigrain Jacobi-Davidson code on the three matrices using the parameters given in Section 4.1. The tests were run on collections of homogeneous machines (Clusters A and B), SMP's (Cluster C), and heterogeneous collections (Clusters A, B, and C together). The fine-grain tests show the scalability of the application using only one solve group with $k = 1, 4$. The multigrain experiments use $k = 4$ and two or four solve groups to solve the four correction equations at each iteration. We construct the solve groups so that machines within the same solve group are homogeneous.

Hierarchical partitionings

Configuration	$A_{16}A_{16}C_{16}C_{16}$ - 64 processors				$A_{32}B_{32}C_{32}C_{32}$ - 128 processors			
Matrix	Gather Scatter	MG inner solve	MG Total	FG	Gather Scatter	MG inner solve	MG Total	FG
FL3D268	0.264	3.00	3.26	5.28	0.092	2.12	2.21	5.27
CFD2	0.034	2.53	2.56	3.61	0.035	1.48	1.51	3.11
NASASRB	0.021	1.56	1.58	1.81	0.012	0.98	0.99	1.60

Figure 6: Timings for one correction phase for fine-grain (FG) and multigrain (MG Total) Jacobi-Davidson, using the method of hierarchical partitionings on 64 and 128 processors, with 4 solve groups and $k = 4$.

The first set of tests given in Figure 7 shows that the scalability of the fine-grain implementation decreases significantly for each matrix when scaling from 32 to 64 processors. The multigrain algorithm is competitive with the fine-grain implementation in the case of FL3D268 and CFD2, showing an improvement of 13-17% over FG1 and FG4. The tests performed with NASASRB show that MG2 is only competitive for 32 or more processors and MG4 is only competitive with 64 processors. MG2 and MG4 show performance degradation in the case of NASASRB for 16 and 32 processors because on average only four inner iterations are required per block vector to satisfy inequality (2). We discuss this further in Section 5.

Similar tests were performed on the cluster of dual processor SMP Ultra 60's (Figure 8). With all methods, the speedup is greatly reduced when scaling from 32 to 64 processors. We believe this is due to contention for the network interface card when two processes are running on each machine. As a result, only the 64 node tests for FL3D268 and CFD2 show a significant performance improvement ($16 - 22\%$) over the fine-grain tests.

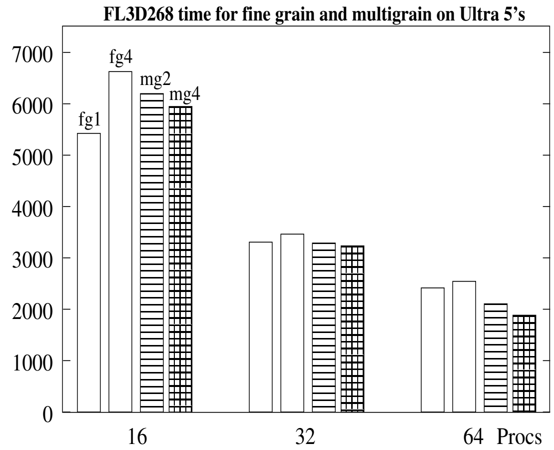
The final set of tests on the SciClone combine the Ultra 5's and Ultra 60's and are given in Figure 9. The results obtained by combining 32 Ultra 5's and 32 Ultra 60's are similar to those shown above for 64 Ultra 5's. Multigrain is able to overcome latencies and provide further improvements in run time. For each test matrix, P_{opt} is between 32 and 64 processors on the SciClone. Multigrain successfully extends P_{max} by a factor of $k = 4$. Furthermore, the SciClone results show that the multigrain code achieves the smallest run time in each of the three tests.

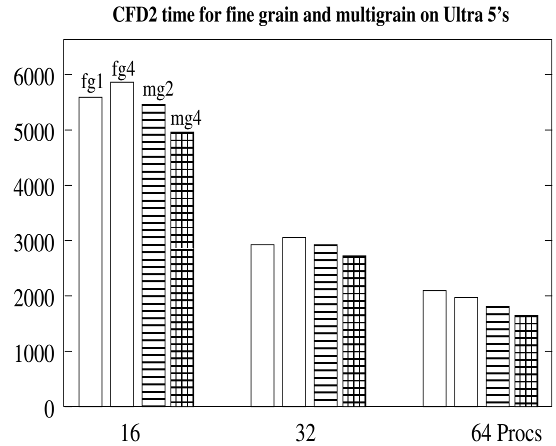
4.3 IBM SP tests

We perform some tests on an IBM SP at the National Energy Research Scientific Computing Center (NERSC). The SP consists of 184 compute nodes each with 16 375MHz processors, 8MB of level 2 cache per processor, 16-64GB of memory, and two network switch interface cards. To reduce network traffic, we configured all processes on a node to belong to the same solve group and utilize shared memory MPI. Four solve groups were created for each of the multigrain tests.

The test results obtained for FL3D268 and CFD2 are given in Figure 10. The fine-grain tests show improvements up to 128 processors. The multigrain code accomplishes the goal of extending the speedup by a factor of the block size. It reduces the effects of latency and achieves up to 60% lower run time compared to the fine-grain implementation.

	Ultra 5 Solve Group Configurations	
P	Fine grain (fg1,4)	Multigrain (mg2,4)
16	A_{16}	$A_8 A_8$ $A_4 A_4 A_4 A_4$
32	A_{32}	$A_{16} A_{16}$ $A_8 A_8 A_8 A_8$
64	$(A_{32} B_{32})$	$A_{32} B_{32}$ $A_{16} A_{16} A_{16} A_{16}$





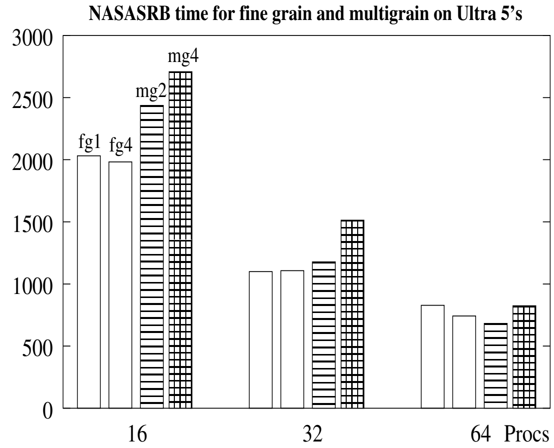
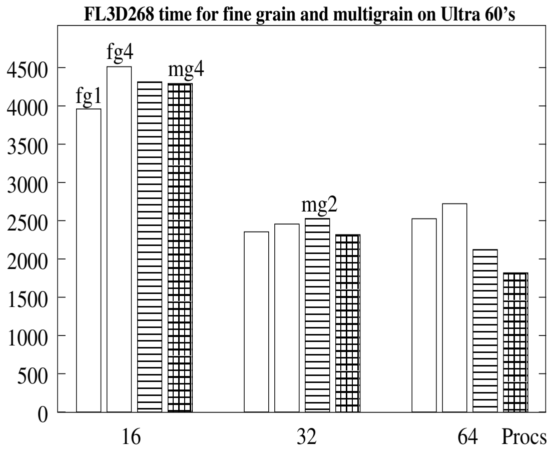
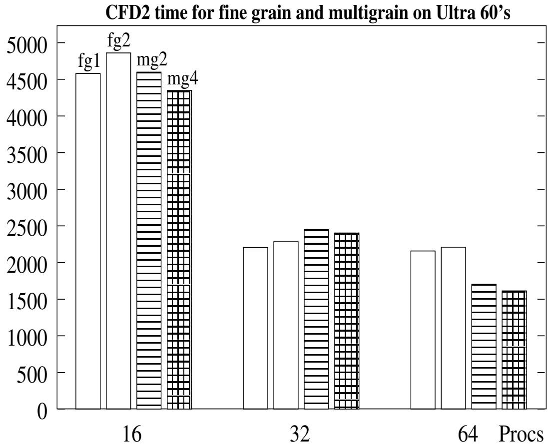


Figure 7: Run times of the fine-grain and multigrain Jacobi-Davidson implementations on the Ultra 5's for each matrix tested with block size $k = 4$. The bar charts measure run time in seconds for various numbers of processors. For each number of processors, there is a string representing the node configuration used. The letters indicate which subcluster was used, and the subscripts indicate the number of processors used in that subcluster. For the fine-grain experiments, there is only one solve group. For multigrain experiments, each letter and subscript combination represents a solve group. For those multigrain experiments with only two letter descriptions, two solve groups were used to solve the four correction equations. Letters within parenthesis represent processors from different subclusters that are part of the same solve group. FG1 and FG4 indicate fine-grain tests with $k = 1$ and $k = 4$ while MG2 and MG4 indicate the tests with two and four solve groups with $k = 4$.

	Ultra 60 Solve Group Configurations	
P	Fine grain (fg1,4)	Multigrain (mg2,4)
16	$C16$	$C8C8$ $C4C4C4C4$
32	$C32$	$C16C16$ $C8C8C8C8$
64	$C64$	$C32C32$ $C16C16C16C16$





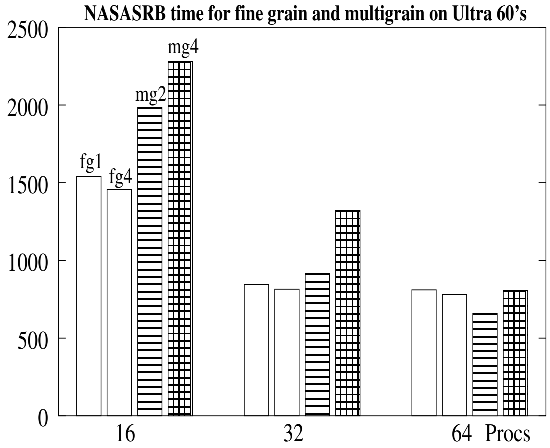
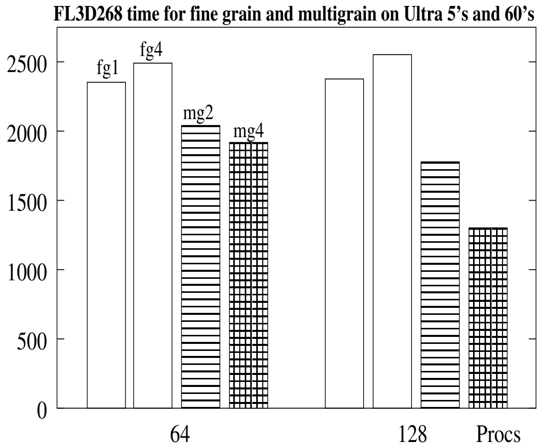
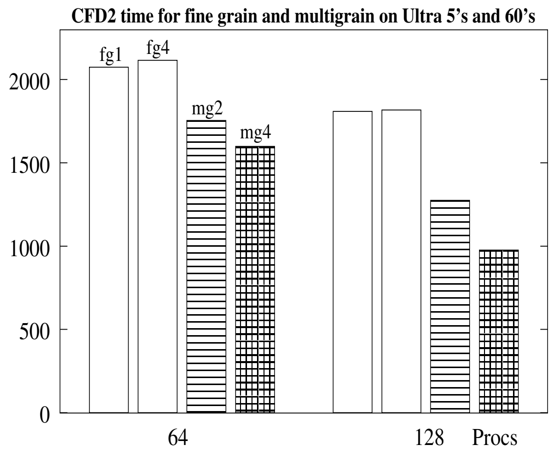


Figure 8: Comparison of the fine-grain and multigrain implementations on the cluster of Ultra 60's. For 16 and 32 processors, only one processor on each machine is utilized. When both processors are used, the scalability of the application suffers due to contention for the network interface.

	Heterogeneous Solve Group Configurations	
P	Fine grain (fg1,4)	Multigrain (mg2,4)
64	$(A32C32)$	$A32C32$ $A16A16C16C16$
128	$(A32B32C64)$	$(A32B32)C64$ $A32B32C32C32$





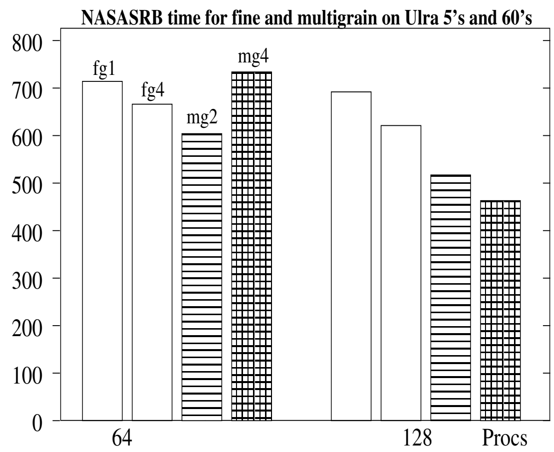


Figure 9: Comparison of the fine-grain and multigrain implementations by combining the Ultra 5's and Ultra 60's. There is little or no improvement achieved by the fine-grain code when scaling from 64 to 128 processors. Multigrain does perform significantly better though. We expect it would continue to show improvements for more than 128 processors.

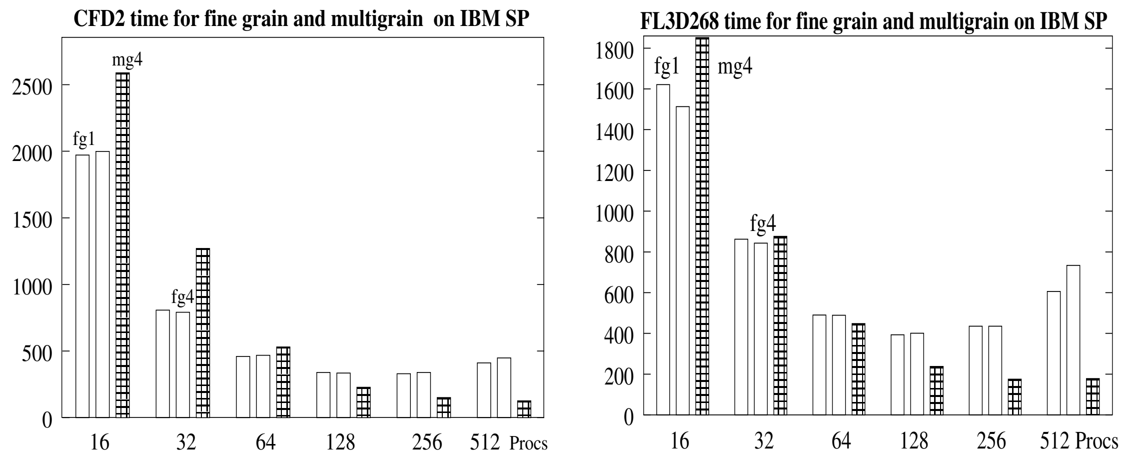


Figure 10: Comparison of fine-grain and multigrain Jacobi-Davidson on an IBM SP. FG1 and FG4 indicate the run time of the fine grain code with $k = 1$ and $k = 4$ respectively. The multigrain tests were run with four solve groups and $k = 4$.

5 Performance model

We now present a performance model of the correction phase to further explain our test results and to predict when the multigrain algorithm should be used. The fine-grain model depends on the parameters: N, P, k, m, $T_A(P)$, $T_M(P)$, T_{blas1}, $T_{allreduce}(P)$, and $T_{all-to-all}(P,k)$. The parameters $T_A(P)$, $T_M(P)$, and T_{blas1} are the times to perform one matrix-vector multiply, one preconditioner application, and one floating-point operation during a BLAS dot product or daxpy respectively. The parameters $T_{allreduce}(P)$, and $T_{all-to-all}(P,k)$ are the times for MPI allreduce and all-to-all as a function of processors and block size. All the parameters except $T_A(P)$ and $T_M(P)$ can be measured once for various P values on some machine and used in later experiments. Because the matrix-vector multiply and preconditioning operators are user defined, the values of $T_A(P)$ and $T_M(P)$ must be measured for each particular problem. Note that for multigrain, P should be the number of processors in the solve group.

Based on these variables, we obtain the time for ddot and daxpy operations as:

$$T_{ddot}(P) = 2 * \frac{N}{P} T_{blas1} + T_{allreduce}(P), \quad \text{and} \quad T_{daxpy}(P) = 2 * \frac{N}{P} * T_{blas1}.$$

The correction equation can be modeled as an initialization step and m steps of BCGSTAB. In each BCGSTAB step, besides some vector updates and dot products, two matrix vector multiplication and preconditioning operations are needed. All-to-all times are not included in fine grain.

$$\begin{aligned} T_{cor}(P) \quad = \quad & 2 * T_M(P) + T_{daxpy}(P) + 4 * T_{ddot}(P) + \\ & 8 * m * T_{ddot}(P) + 12 * m * T_{daxpy}(P) + 2 * m * T_A(P) + 2 * m * T_M(P) + \\ & 3 * T_{all-to-all}(P,k) \end{aligned}$$

To test the model, we forced BCGSTAB to perform 10 iterations per block vector during each correction phase with matrix NASASRB. The eigensolver was allowed to run for several iterations and the time to

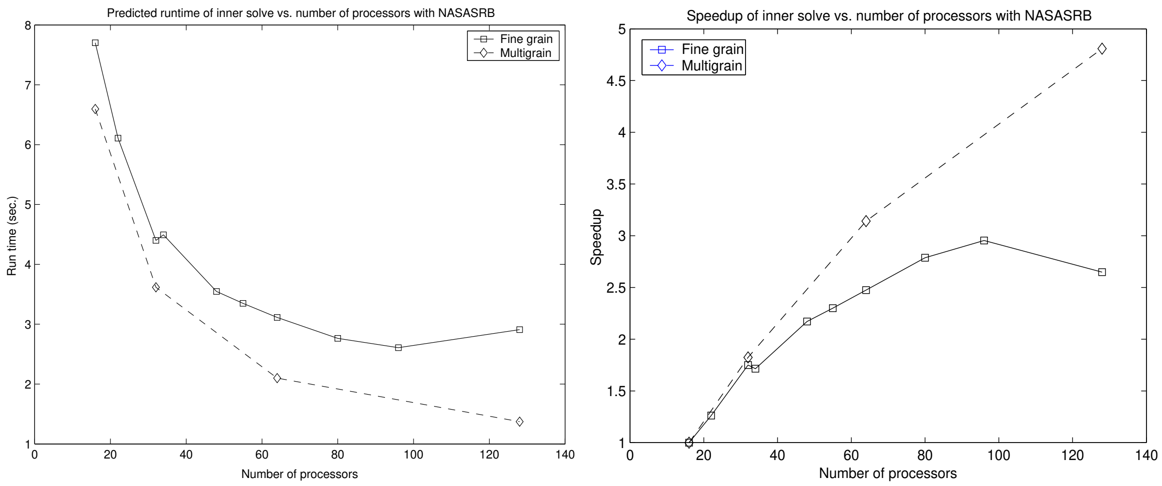


Figure 11: Model predictions for run time and corresponding speedup of the fine grain and multigrain correction phases when 10 iterations of BCGSTAB are applied per block vector. The predictions were made with $k = 4$ and four solve groups and the speedups are with respect to the predicted run time achieved with 16 processors.

complete the correction phase was averaged over the iterations. Furthermore, tests run on 64 or fewer nodes were run strictly on clusters A and B. The model was able to predict the length of the correction phase to within 5% for all fine grain and multigrain tests except the multigrain case for 128 processors which was 10% higher than the experimental time. This may be related to the fact that cluster C consists of faster machines which the model does not account for.

In Figure 11 we show the run times and speedup given by the model's predications for NASASRB. The model shows that multigrain does indeed extend the speedup of the correction phase. Of further interest is the fact that multigrain outperforms the fine grain implementation for the four configurations: $A_4A_4A_4A_4$, $A_8A_8A_8A_8$, $A_{16}A_{16}B_{16}B_{16}$, and $A_{32}B_{32}C_{32}C_{32}$. However, we do not see commensurate gains in the previous experiments when solving for the 50 lowest eigenvalues. This is because the quality of the preconditioner was high, so on average only four inner iterations instead of ten per block vector were needed to obtain convergence. Furthermore, not all processors converged in the same number of steps during each outer iteration, resulting in large load imbalances. Obviously, multigrain is a powerful technique but it requires problems that need many inner iterations.

6 Conclusions

We have proposed multigrain, a latency-tolerant modification of parallel, block iterative methods that perform preconditioning using an inner iterative method. Multigrain employs fine-grain parallelism during the projection phase and coarse-grain parallelism during the preconditioning phase. This is a reasonable application of coarse-grain parallelism when solving difficult problems that require many inner iterations because most of the execution time and communication is attributed to the inner solver.

In the case of the Jacobi-Davidson method, we can induce coarse granularity during the correction phase by splitting the processors into subgroups and having each subgroup solve a distinct correction equation.

	Fine grain $k=1$			Fine grain $k=4$		
Configuration	Time	Matvecs	Iterations	Time	Matvecs	Iterations
A_{16}	5424	63210	1745	6627	74560	647
A_{32}	3306	63468	1740	3463	69944	621
$A_{32}B_{32}$	2417	64184	1763	2544	71170	629
C_{16}	3961	63210	1745	4512	74560	647
C_{32}	2355	63468	1740	2457	69944	621
C_{64}	2526	64184	1763	2722	71170	629
$A_{32}C_{32}$	2353	64184	1763	2491	71170	629
$A_{32}B_{32}C_{64}$	2377	64939	1778	2552	71852	628

	Multigrain $k=4$		
Configuration	Time	Matvecs	Iterations
A_8A_8	6197	70716	624
$A_4A_4A_4A_4$	5948	71180	627
$A_{16}A_{16}$	3288	70536	623
$A_8A_8A_8A_8$	3234	71798	633
$A_{32}B_{32}$	2107	71890	628
$A_{16}A_{16}B_{16}B_{16}$	1885	73356	641
C_8C_8	4315	70716	624
$C_4C_4C_4C_4$	4293	71180	627
$C_{16}C_{16}$	2529	70536	623
$C_8C_8C_8C_8$	2318	71798	633
$C_{32}C_{32}$	2122	71890	628
$C_{16}C_{16}C_{16}C_{16}$	1819	73356	641
$A_{32}C_{32}$	2041	71890	628
$A_{16}A_{16}C_{16}C_{16}$	1918	73356	641
$(A_{32}B_{32})C_{64}$	1778	75124	649
$A_{32}B_{32}C_{32}C_{32}$	1302	70992	623

Figure 12: Fine grain and multigrain results for FL3D268 on SciClone

Since each solve group can be associated with a physically different cluster, computation occurs locally at a much smaller latency rate, and with minimal intercluster communication. Our experiments with a multigrain Jacobi-Davidson code show that the multigrain parallelism extended the scalability of the solver both on a collection of clusters and an MPP composed of SMP's.

Multigrain should be applied only when fine grain starts to face scalability problems. We have constructed a model that can be used to predict how much time is required by the fine and coarse-grain Jacobi-Davidson correction phase. The model serves as an analytical tool for determining why and under what conditions the multigrain method is effective.

	Fine grain $k = 1$			Fine grain $k = 4$		
Configuration	Time	Matvecs	Iterations	Time	Matvecs	Iterations
A_{16}	5590	43261	2074	5863	47188	574
A_{32}	2924	43345	2078	3056	47354	576
$A_{32}B_{32}$	1933	43742	2097	1974	47280	575
C_{16}	4579	43261	2074	4861	47188	574
C_{32}	2207	43345	2078	2284	47354	576
C_{64}	2158	43742	2097	2209	47280	575
$A_{32}C_{32}$	2074	43742	2097	2116	47280	575
$A_{32}B_{32}C_{64}$	1809	44531	2134	1817	47532	578

	Multigrain $k = 4$		
Configuration	Time	Matvecs	Iterations
A_8A_8	5458	47020	572
$A_4A_4A_4A_4$	4963	47166	574
$A_{16}A_{16}$	2925	46348	564
$A_8A_8A_8A_8$	2725	48952	595
$A_{32}B_{32}$	1814	47522	578
$A_{16}A_{16}B_{16}B_{16}$	1652	46768	569
C_8C_8	4598	47020	572
$C_4C_4C_4C_4$	4349	47166	574
$C_{16}C_{16}$	2451	46348	564
$C_8C_8C_8C_8$	2403	48952	595
$C_{32}C_{32}$	1704	47522	578
$C_{16}C_{16}C_{16}C_{16}$	1612	46768	569
$A_{32}C_{32}$	1754	47522	578
$A_{16}A_{16}C_{16}C_{16}$	1599	46768	569
$(A_{32}B_{32})C_{64}$	1275	47112	573
$A_{32}B_{32}C_{32}C_{32}$	976	47942	583

Figure 13: Fine grain and multigrain results for CFD2 on SciClone

	Fine grain $k = 1$			Fine grain $k = 4$		
Configuration	Time	Matvecs	Iterations	Time	Matvecs	Iterations
A_{16}	2031	16565	1831	1982	16636	449
A_{32}	1100	16021	1764	1107	17032	443
$A_{32}B_{32}$	828	16326	1868	742	16916	509
C_{16}	1539	16565	1831	1455	16636	449
C_{32}	844	16021	1764	815	17032	443
C_{64}	810	16326	1868	780	16916	509
$A_{32}C_{32}$	714	16326	1868	666	16916	509
$A_{32}B_{32}C_{64}$	692	15301	1724	621	16288	422

	Multigrain $k = 4$		
Configuration	Time	Matvecs	Iterations
A_8A_8	2437	17810	509
$A_4A_4A_4A_4$	2708	16934	471
$A_{16}A_{16}$	1178	17076	467
$A_8A_8A_8A_8$	1513	18298	495
$A_{32}B_{32}$	682	16184	398
$A_{16}A_{16}B_{16}B_{16}$	824	17688	543
C_8C_8	1983	17810	509
$C_4C_4C_4C_4$	2281	16934	471
$C_{16}C_{16}$	916	17076	467
$C_8C_8C_8C_8$	1323	18298	495
$C_{32}C_{32}$	657	16184	398
$C_{16}C_{16}C_{16}C_{16}$	806	17688	543
$A_{32}C_{32}$	604	16184	398
$A_{16}A_{16}C_{16}C_{16}$	734	17688	543
$(A_{32}B_{32})C_{64}$	517	16836	483
$A_{32}B_{32}C_{32}C_{32}$	463	15934	422

Figure 14: Fine grain and multigrain results for NASASRB on SciClone

References

[1] RS/6000 SP: SP Switch and SP Switch2 Performance. Technical report, IBM Corporation, June 2001.

[2] Farazdel A., Archondo-Callao G.R., and Hocks E. et. al. Understanding and Using the SP Switch. Technical Report SG24-5161-00, IBM Corporation, 1999.

[3] C. Aykanat, F.Ozguner, and D.S. Scott. Vectorization and parallelization of the conjugate gradient algorithm on hypercube-connected vector processors. *Microprocessing & Microprogramming*, 29:67–82, Sept 1990.

[4] L. Bergamaschi, G. Pini, and F. Sartoretto. Parallel preconditioning of a sparse eigensolver. *Parallel Computing*, 27(7):963–76, 2001.

[5] Edmond Chow. ParaSails: Parallel sparse approximate inverse (least-squares) preconditioner. Technical report, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, L-560, Box 808, Livermore, CA 94551, 2001.

[6] J. Cullum and R. A. Willoughby. *Lanczos algorithms for large symmetric eigenvalue computations*, volume 1: Theory of *Progress in Scientific Computing; v. 3*. Birkhauser, Boston, 1985.

[7] Tim Davis. University of florida sparse matrix collection. *NA Digest*, 97(23), June 1997.

[8] E. de Sturler and H.A. Van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Applied Numerical Mathematics*, 18(4):441–59, Oct 1995.

[9] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H.A. van der Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, Philadelphia, PA, 1998.

[10] D. R. Fokkema, G. L. G. Sleijpen, and H. A. van der Vorst. Jacobi-Davidson style QR and QZ algorithms for the partial reduction of matrix pencils. *SIAM J. Sci. Comput.*, 20(1), 1998.

[11] I. Foster and C. Kesselman, editors. *The Grid — Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[12] G. H. Golub and R. Underwood. The block Lanczos method for computing eigenvalues. In J. R. Rice, editor, *Mathematical Software III*, pages 361–377, New York, 1977. Academic Press.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 340 Pine Street, San Francisco, CA 94104-3205, 2 edition, 1996.

[14] K. Hwang and Z. Xu. *Scalable Parallel Computing*. WCB/McGraw Hill, 1998.

[15] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–85, 1998.

[16] J. R. McCombs, R. T. Mills, and A. Stathopoulos. Dynamic load balancing of an iterative eigensolver on networks of heterogeneous clusters. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, to appear, 2003.

[17] J. R. McCombs and A. Stathopoulos. Multigrain parallelism for eigenvalue computations on networks of clusters. In *Proceedings of the Eleventh IEEE International Symposium On High Performance Distributed Computing*, pages 143–149, Los Alamitos, California, 2002. IEEE.

[18] Dianne P. O'Leary. The block conjugate gredient algorithm and related methods. *Lin. Alg. Appl.*, 29:293–322, February 1980.

[19] Dianne P. O'Leary. Parallel implementation of the block conjugate gradient algorithm. *Parallel Computing*, 5:127–139, 1987.

[20] Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM, Philadelphia, PA, 1998.

[21] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[22] Y. Saad and K. Wu. Parallel SPARSe matrix LIBrary (P_SPARSLIB): the iterative solvers module. Technical Report 94-008, Army High Performance Computing Research Center, Minneapolis, 1994.

[23] Yosef Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2.

[24] Yousef Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, March 1993.

[25] Yousef Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, 1996.

[26] V. Simoncini and E. Gallopoulos. A hybrid block gmres method for nonsymmetric systems with multiple right-hand sides. *Journal of Computational and Applied Mathematics*, 66:457–69, Jan 1996.

[27] G. L. G. Sleijpen, A. G. L. Booten, D. R. Fokkema, and H. A. van der Vorst. Jacobi-davidson type methods for generalized eigenproblems and polynomial eigenproblems. *BIT*, 36(3):595–633, 1996.

[28] Gerald L.G. Sleijpen and Henk A. van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM Journal of Matrix Analysis and Applications*, 17:401–425, 1996.

[29] A. Stathopoulos and C. F. Fischer. Reducing synchronization on the parallel Davidson method for the large,sparse, eigenvalue problem. In *Supercomputing '93*, pages 172–180, Los Alamitos, CA, 1993. IEEE Comput. Soc. Press.

[30] A. Stathopoulos and J. R. McCombs. A parallel, block, Jacobi-Davidson implementation for solving large eigenproblems on coarse grain environments. In *1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2920–2926. CSREA Press, 1999.